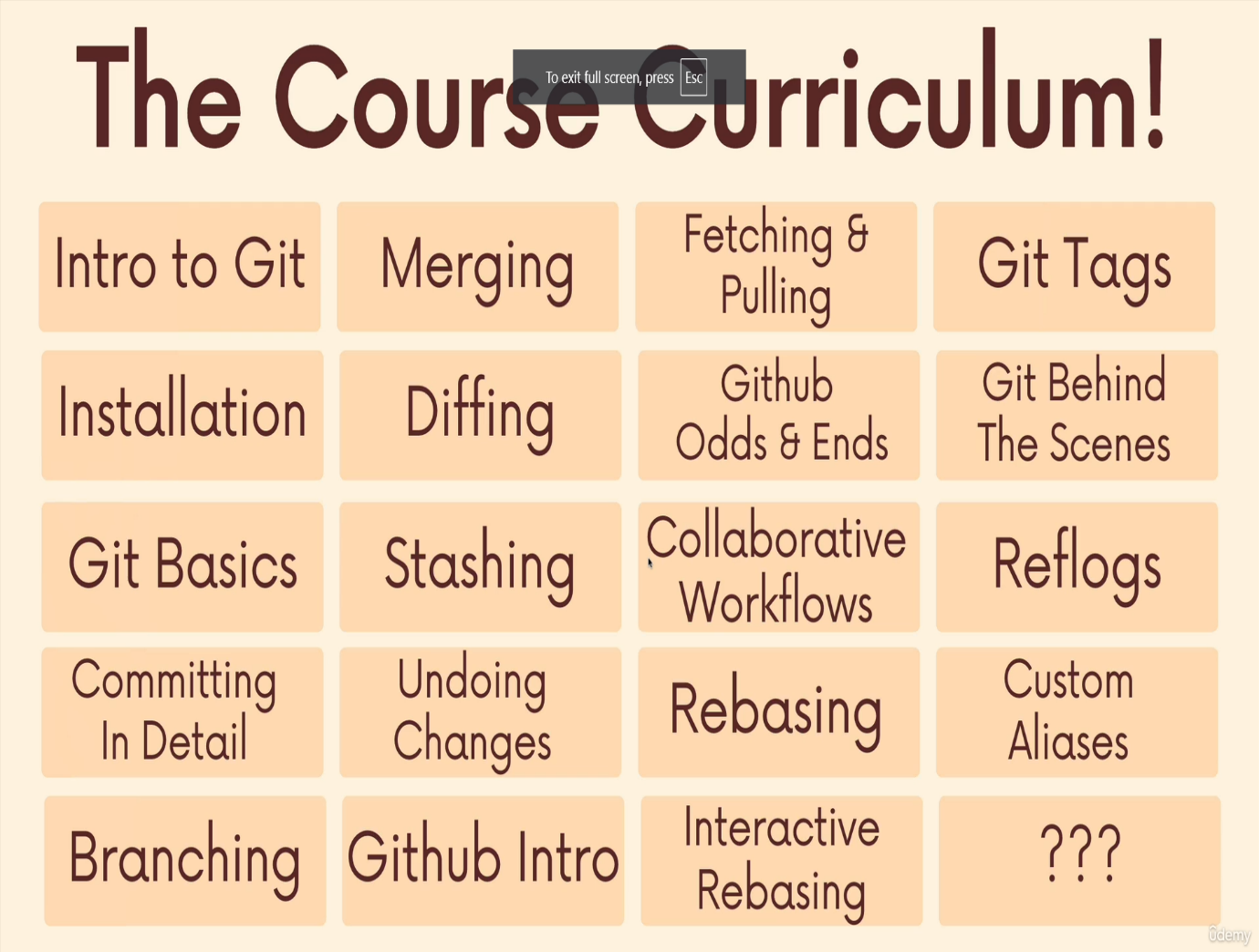
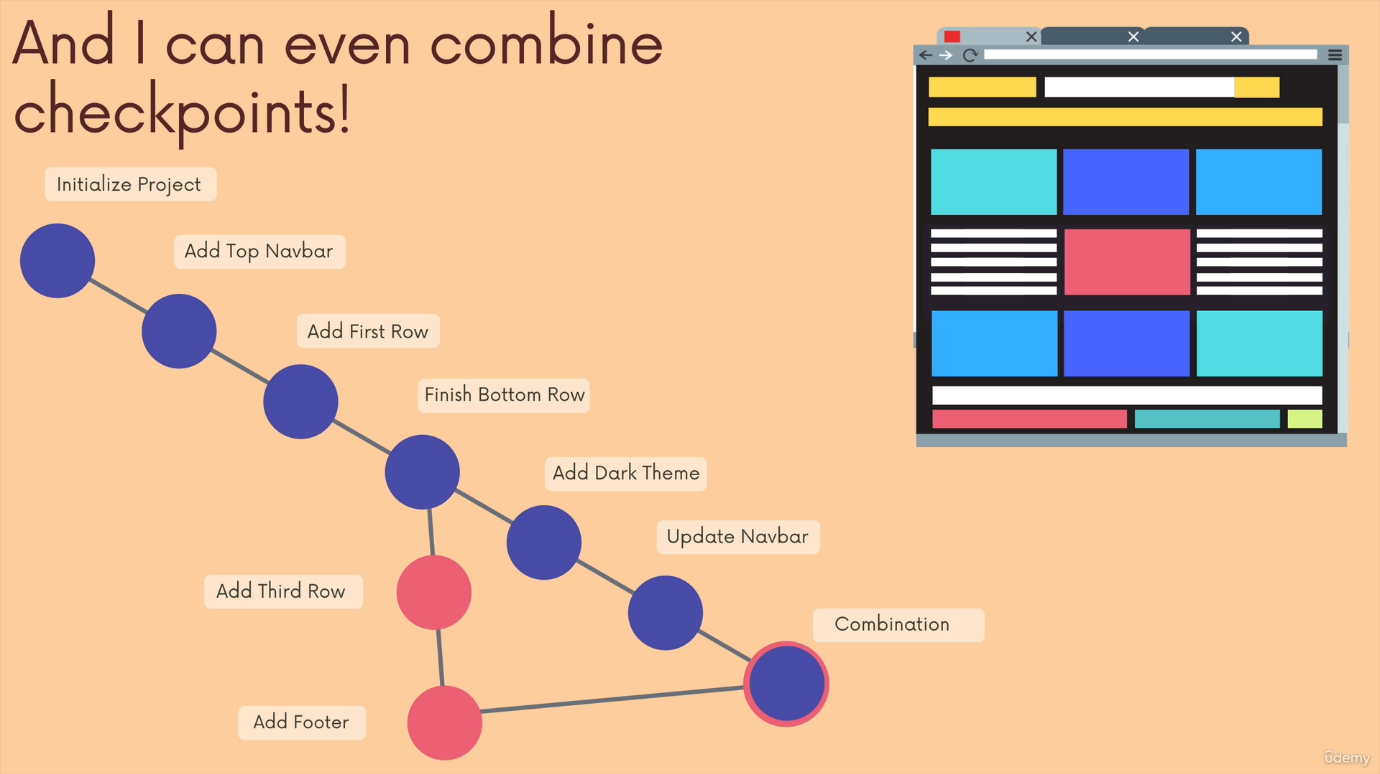
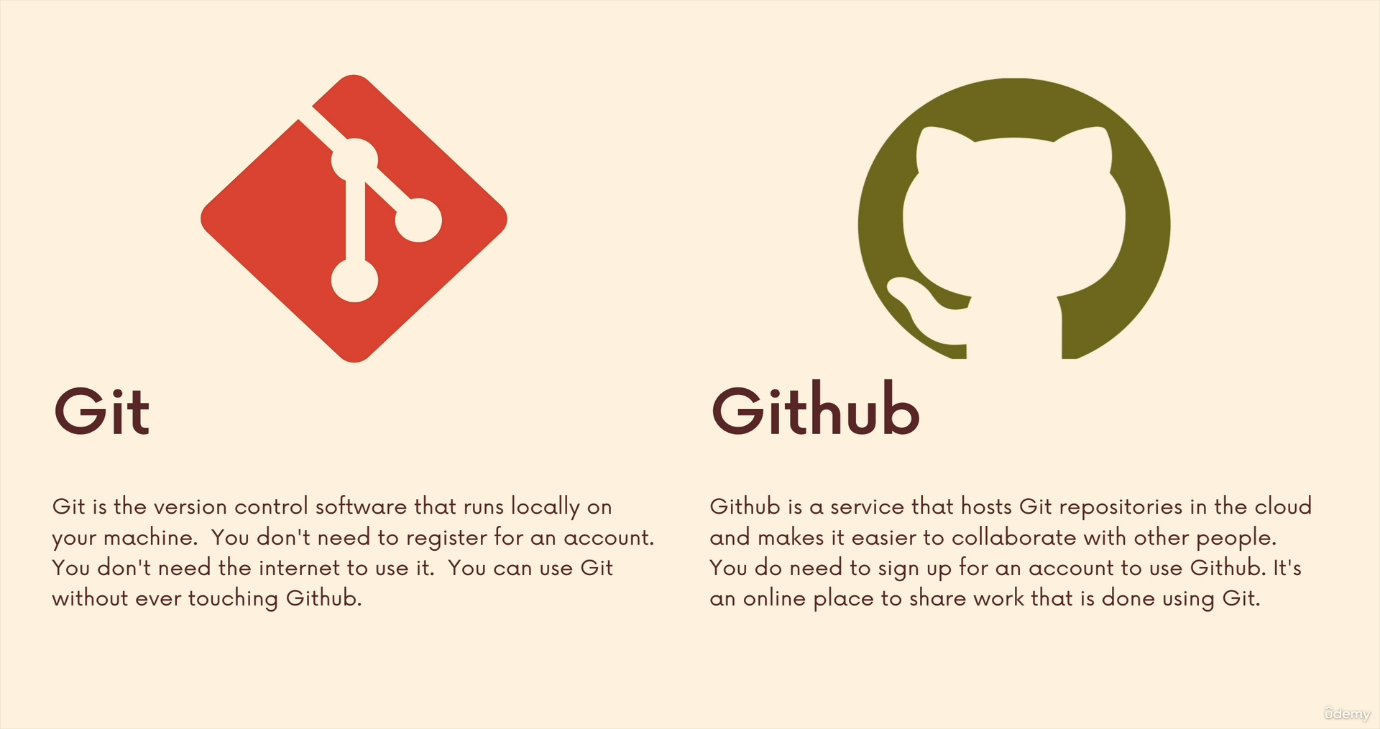
**Section 1 : Course Orientation**



**Section 2 : Introducing Git!**

* Git is the world's most popular version control system
* Version Control is software that tracks and manages changes to files over time
* Version control systems generally allow users to revisit earlier versions of the files, compare changes between versions, undo changes, and a whole lot more.
* Git helps us…
  + Track changes across multiple files
  + Compare versions of a project
  + “Time travel” back to old versions
  + Revert to a previous version
  + Collaborate and share changes
  + Combine changes
* 
* Linus Torvalds invented Git and GIT stands for Global Information Tracker
* SCM stands for Source Control Managment
* 

**Section 3: Installation & Setup**

* Git was created as a command-line tool. To use it, we run various git commands in a Unix shell. This is not the most user-friendly experience, but it’s at the very core of Git!
* Over the last few years, companies have created graphical user interfaces for Git that allow people to use Git without having to be a command-line expert. Popular Git GUI’s include:
  + GitHub Desktop
  + SourceTree
  + Tower
  + GitKraken
  + Ungit
* GUI Clients

Pros:

* Way lower barrier-of-entry for beginners compared to the command-line.
* Friendlier to use. Can be a much better experience (when it works).
* Some people prefer the visual experience, even those who can use the command-line.

Cons:

* At times, there is lots of “magic” involved. The inner-workings of Git are obfuscated and hidden away with GUIs.
* Often leads to dependence on a particular piece of software.
* When things go seriously wrong, it can be very challenging to fix without the command-line.
* The interfaces (UI/UX) and options vary between different GUIs.
* The Command Line

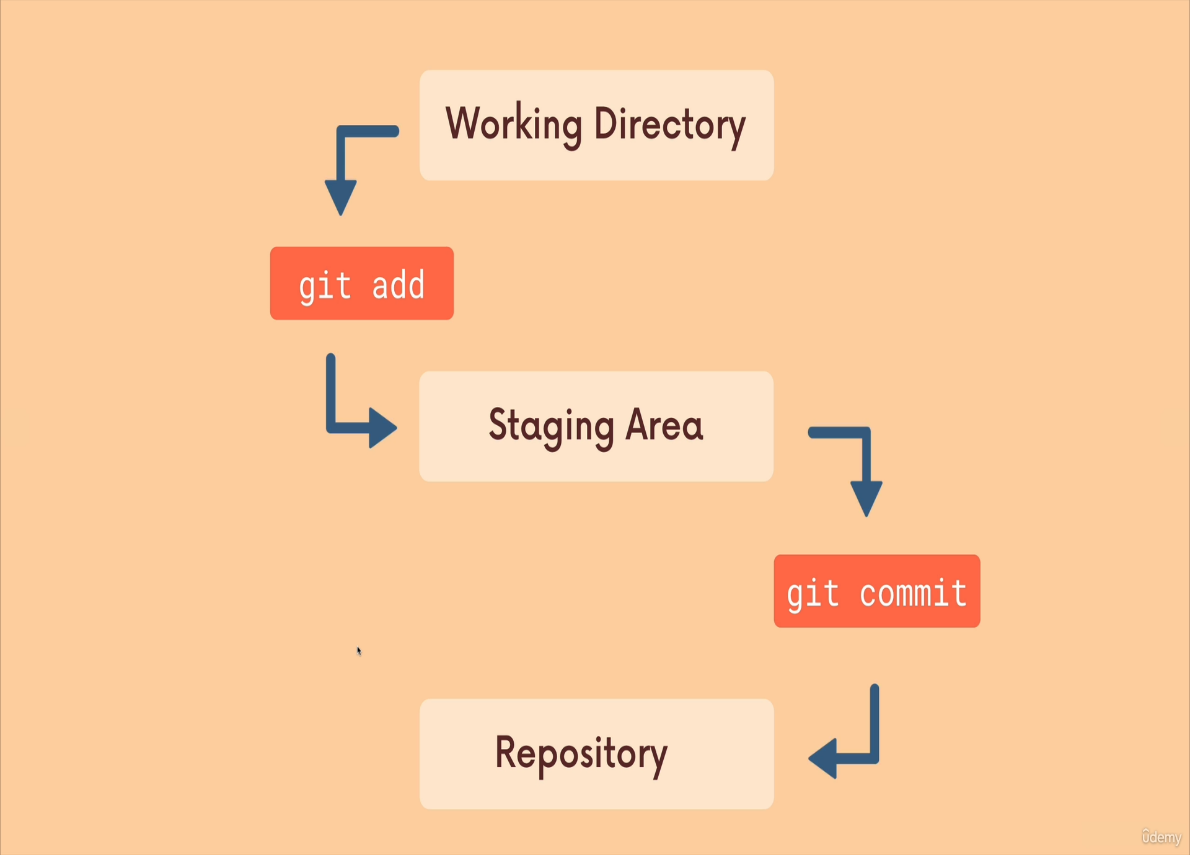
Pros:

* Git is a command-line tool. All the documentation and resources online will refer to the command-line way faster once you get comfortable with it!
* Some of the advanced Git features are only available on the command-line.
* The commands are always the same no matter what machine you are on!

Cons:

* Not beginner-friendly. At all. Can be difficult to learn and remember not just a command at first.
* Even for some practiced users, the command-line interface is just not a good experience. It’s really a matter of preference.
* Configuring git such that we can see our name and email in logs when several users collaborated
  + git config --global user.name "Bhargav"
  + git config --global user.email bhargavs3499@gmail.com
* Terminal Crash course on linux commands
  + Start .(windows)/open .(macos) will open the current folder
  + Rm to remove files
  + Rm -rf to remove folders/directories (r-recursive f-force)
  + Ls -a will show me the actual files and hidden files as well (a-all)

**Section 4: The Very Basics Of Git: Adding & Committing:**

* A Git "Repo" is a workspace which tracks and manages files within the folders.
* The **git init** is used to create a new git repository. Before we can do anything git -related ,we must initialize a repo first. This is something you do once per project. Initialize the repo in the top-level folder containing your project.(**.git is created and hidden**)
* The **git status** command shows the current state of the working directory and staging area, listing changes that are staged, unstaged, and untracked
* The **.git** directory is the heart of a Git repository. It contains all the metadata and object database for the repository, including:
  + Commit history: All the commits made in the repository.
  + Configuration files: Settings and preferences for the repository.
  + References: Information about branches, tags, and remotes.
  + Staging area: The index where changes are added before committing.
* Essentially, it tracks all changes and enables version control functionalities. If you delete this directory, you lose the entire version history and configuration of the repository.
* 
* If you made this mistake delete .git directory using rm -rf .git
* 

* We use the **git add** command to stage changes to be committed ("It’s a way of telling Git ,please include this change in our next commit")
* We use the git commit command to actually commit changes from the staging area. When making a commit, we need to provide a commit a message that summarizes the changes and work snapshotted in the commit
* Running **git commit** will commit all staged changes. It also opens up a text editor and prompts you for a commit message. This can be overwhelming when you're starting out, so instead you can use … (**git commit**)
* The -m flag allows us to pass an inline commit message, rather than launching a text editor. We'll learn more about writing good commit messages later on. (**git commit -m "my message"**)
* We use the **git log** command to get the history or logs of git commits
* We use **git add .** to stage all changes at once.

**Section 5: Commits in Detail (And Related Topics)**

**Atomic Commits**

* When possible, a commit should encompass a single feature, change, or fix. In other words, try to keep each commit focused on a single thing. This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review.
* If you are using git commit command and if it opens in vim you need to enter I for insert mode and enter commit message and to quit after that you need to type :wq for saving and exit(w-write and q-quit)
* For configuring git to open instead of vim after git commit or to enter large commit message we can configure vscode as code editor for commit using the command git config --global core.editor "code.wait"(install code command in path in vscode cntrl+shift+k)
* We use the command git log --oneline to log commit history in one line
* Amending Commits
* Suppose you just made a commit and then realized you forgot to include a file. Or, maybe you made a typo in the commit message that you want to correct.
* Rather than making a brand new separate commit, you can ‘redo’ the previous commit using the --amend option.
  + git commit -m 'some commit'
  + git add forgotten.file
  + git commit --amend

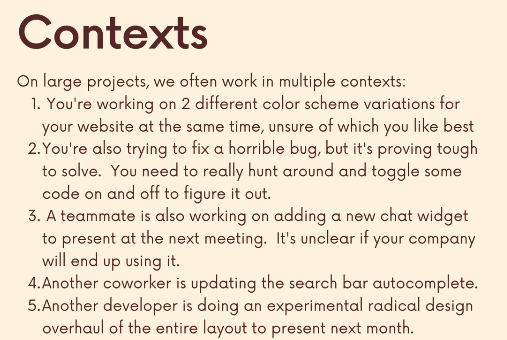
**Ignoring Files**

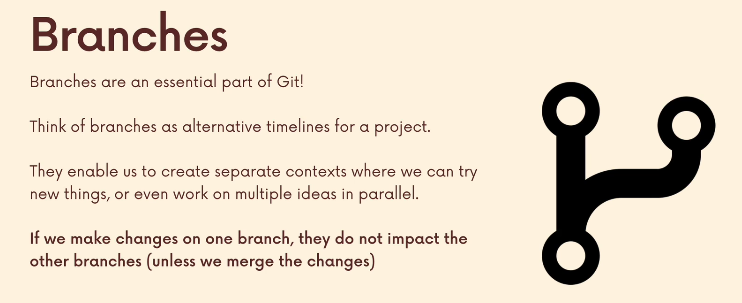
* We can tell Git which files and directories to ignore in a given repository, using a .gitignore file. This is useful for files you know you NEVER want to commit, including:
* Secrets; API keys; credentials, etc.
* Operating system files (.DS\_Store on Mac)
* Dependencies & packages

**.gitignore**

* Create a file called .gitignore in the root of a repository. Inside the file, we can write patterns to tell Git which files & folders to ignore:
* \*.DS\_Store will ignore files named .DS\_Store
* folderName/ will ignore an entire directory
* \*.log will ignore files with a .log extension

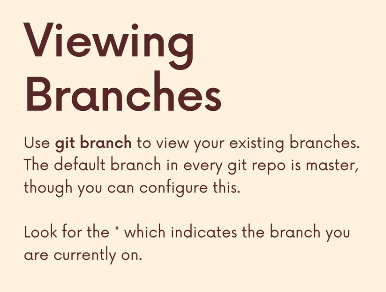
**Section 6:Working with Branches**

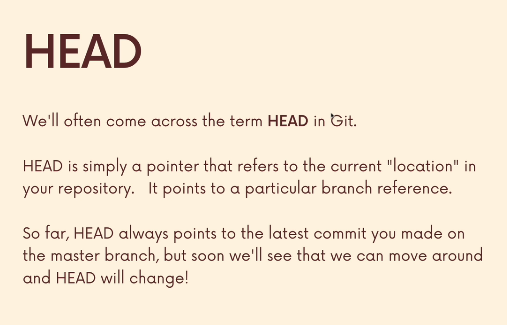
* Each commit has hash attached to it and it contains parent hash attached to it
* 



* 

* 

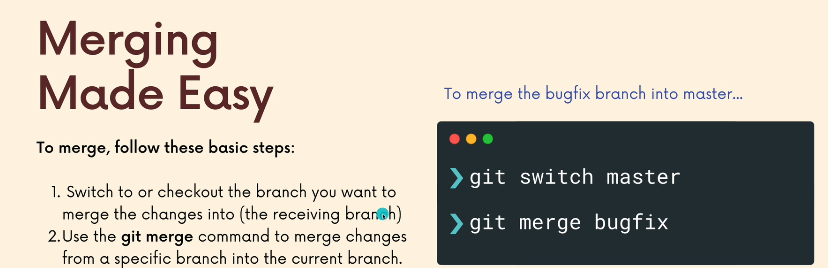
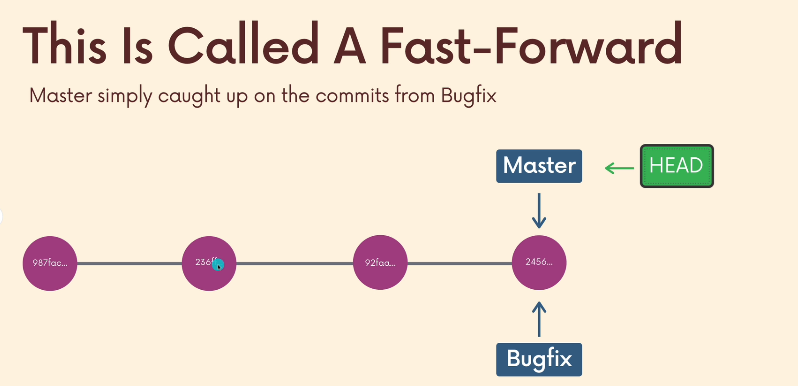
* 
* 

* 
* Switching branches with unstaged changes either will throw an error to commit or if it is a new file then switching the branch takes place but the file will also remains there from the other branch(to avoid this always commit or stash changes before switching out branch)
* For deleting the branch you need to get out of the deleting branch and run **git branch -d branchName**(if it is not in sync with the other branch then you need to force delete using this command **git branch -D branchname**)
* For renaming the branch you need to be in the branch and run **git branch -m newBranchName**
* 
* To view the contents of a file, use the command cat file.
* The .git/HEAD file always points to the current branch's head, typically in the format ref: refs/heads/master.
* The refs/heads directory contains files named after each branch, and these files store the commit IDs for their respective branches.
* **git commit -a -m "message"**: This command commits changes to files that are already tracked by Git. It will not include new files that haven't been staged yet.

**Section 7: Merging Branches ,Oh Boy!**

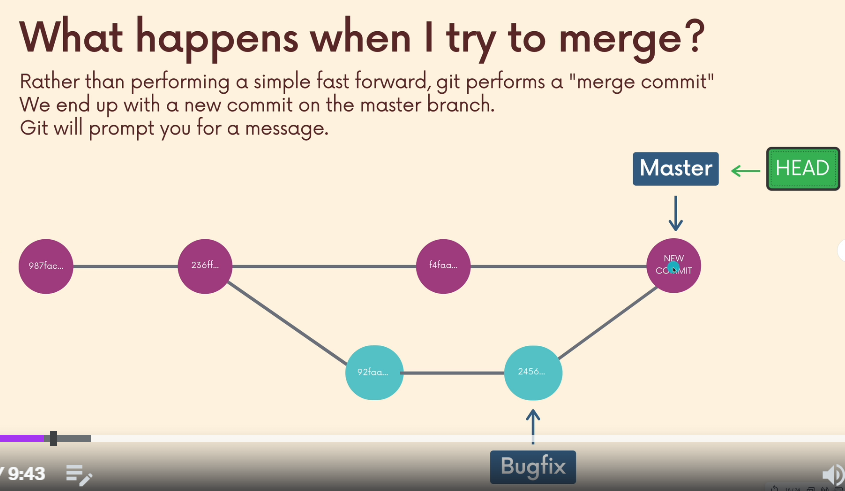
* 
* The main branch is called **master branch** and the other branches working on some feature is called **feature branch**

**Merging Concepts**

* We merge branches ,not specific commits
* We always merge to the current HEAD branch
* 
* Simple merges are called fast forward merges
* 
* Git merge branchname is the command to perform merging
* Git branch -v shows a summary of all branches in your repository along with the latest commit on each branch.

Ex:

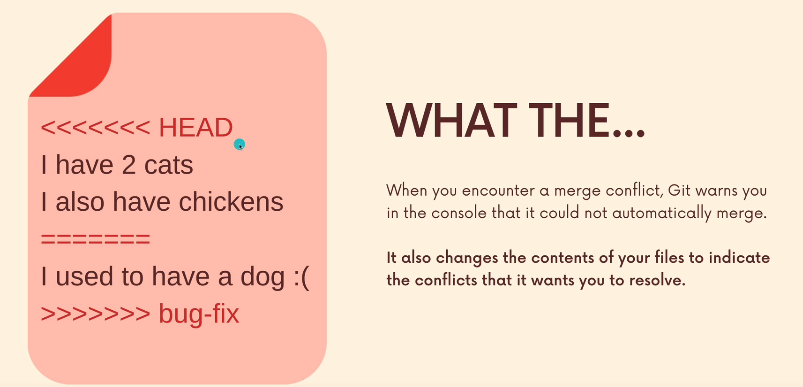
* \* master 2d3ac45 Fix bug in payment module
* feature/xyz 4f2d3b6 Add new feature for user authentication
* develop a1b2c3d Update README file

* 
* Depending on the specific changes you are trying to merge, Git may not be able to automatically merge. This results in merge conflicts, which you need to manually resolve

Ex:

CONFLICT(content):Merge conflict in blah.txt

Automatic merge failed; fix conflicts and then commit the result

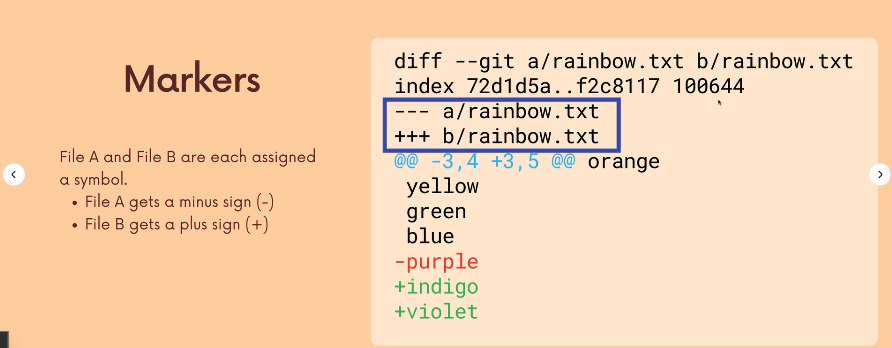


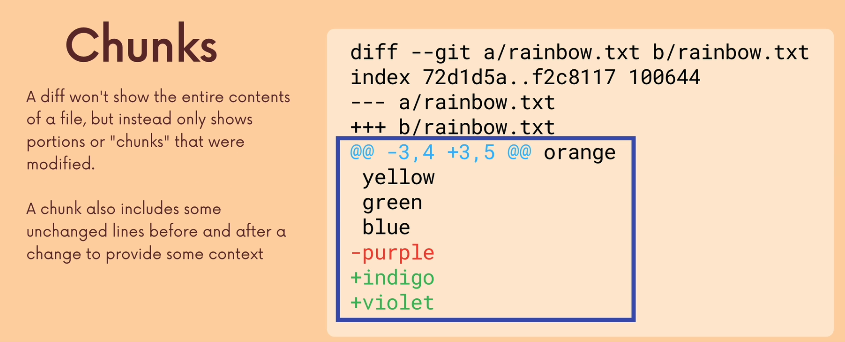


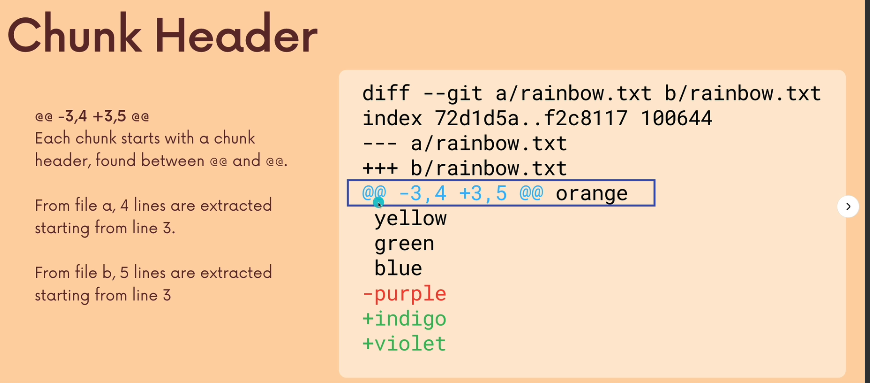
**Section 8: Comparing Changes with Git Diff**

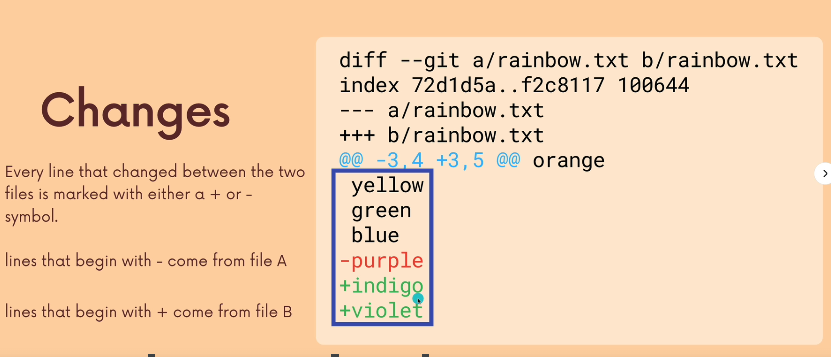




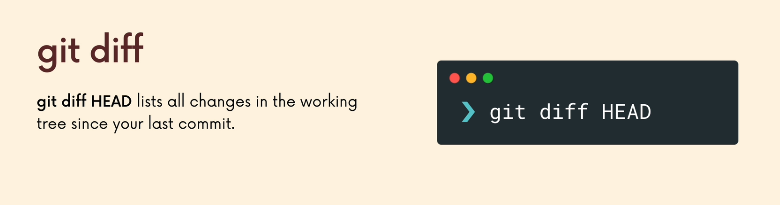








Git commit -am "add orange" to add and commit (this will work if you have atleast added your file to staging area)



Git diff shows all unstaged changes

Git diff HEAD shows all staged and unstaged changes

Git diff --staged or git diff --cached will only show staged changes

Git diff HEAD style/main.css will give you diffing specific changes wrt HEAD

Git diff HEAD style/main.css index.html will give you diffing multiple files wrt HEAD

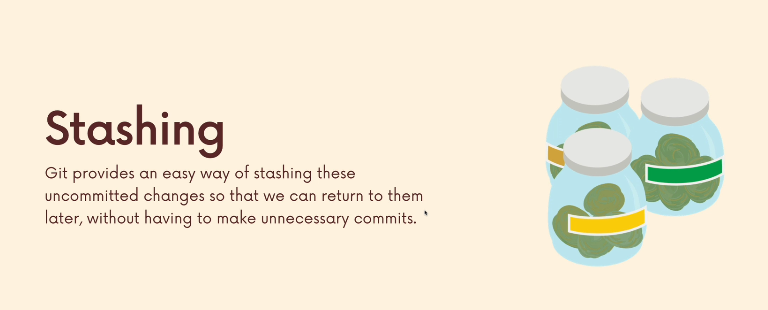
Git diff branch1 branch2 or git diff branch1..branch2 will compare changes between branches

Git diff commit1 commit2 will compare changes across commits

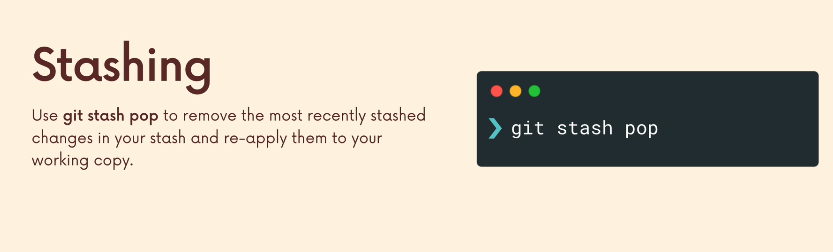
Git diff HEAD HEAD~1 is just a short syntax for "the parent commit of HEAD"

**Section 9: The Ins and Outs of Stashing**

Why we need Git Stash  
1. My changes come with me to the destination branch(while switching branch your changes will come to the branch but not commited to avoid this we use git stash)

1. Git won't let me switch if it detects potential conflicts
2. 







Git stash apply wont remove data in stash whereas in git stash pop the data in stash gets removed and can't be used again

We can use the command git stash list to view the stashes list



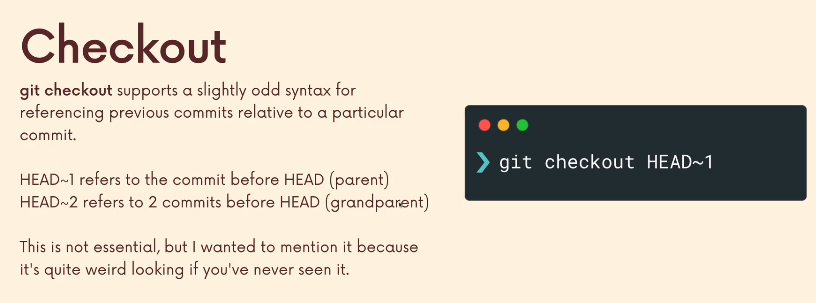
We can apply specific stashes using the command git stash apply stash@{2}





**Section 10: Undoing Changes & Time Travelling**

* We can use Git checkout <commit hash> to view a previous commit. We just need 7 digits(we can get it using git log --oneline) or the whole commit hash(detatched head state)
* Head usually refers to a branch NOT a specific commit
* 

* 
* Git switch - will take me to last branch l was on
* Git checkout HEAD filename (or) git checkout -- filename to restore a file w.r.t to head



